Scala for Professionals

Implicit Fundamentals

Implicit Conversions

Helping Rearrange What Doesn't Fit

As Scala developers, we frequently see syntax that doesn't seem to fit into Scala:

```
// From an sbt build definition
"org.scalatest" %% "scalatest" % "2.2.4" % "test"

// From TimeSpec
"Calling toString" should {
  "return a properly formatted string representation" in {
    Time(9, 30).toString must be === "09:30"
  }
}
```

Both of these examples should produce type errors in the compiler, but somehow they pass.

Implicit Conversions

When the compiler encounters a type error, rather than giving up it looks for an **implicit conversion**.

If the *actual* type doesn't match the *expected* type, Scala looks for an implicit conversion to the expected type:

```
2 - "1" // - expects an Integer... got a String
```

If we try to access a *member* which doesn't exist, Scala looks for an implicit conversion of the **receiver** to something else that has that member:

```
"2" - 1 // String lacks a -(x: Int) method...
```

Defining Implicit Conversions

What *is* an implicit conversion?

An implicit value of unary function type (A => B).

```
implicit val stringToInt : String => Int = ...
```

During compilation, Scala wraps the relevant code in a call to the implicit conversion function

- Since methods are automatically lifted into functions, it's typically idiomatic (not to mention more convenient) to use a method.

Defining Implicit Methods

To define an implicit conversion, we declare a method prefixed with the `implicit` keyword:

```
implicit def stringToInt(s: String): Int =
  Integer parseInt s
```

- While the name of the method doesn't matter, by convention it should be unambiguous and descriptive.
- **Idiomatic Convention**: Implicit conversions should be named `sourceToTarget`, e.g. `stringToInt`

Digression: Implicits Language Import

Since Scala 2.10, the compiler warns us when we define an **implicit method**.†

```
implicit def stringToInt(s: String): Int = Integer.parseInt(s)
// warning: there was one feature warning; re-run with -feature for details

scala> :warnings
/*
<console>:10: warning: implicit conversion method stringToInt should be enabled
by making the implicit value scala.language.implicitConversions visible.
This can be achieved by adding the import clause 'import scala.language.implicitConversions'
or by setting the compiler option -language:implicitConversions.

       implicit def stringToInt(s: String): Int = Integer.parseInt(s)
                    ^
*/
```

† We suppress this warning in our sample project, but in a fresh Scala (non-project) console we can see it.

Digression: Implicits Language Import

Suppressing Implicit Warnings

1. In the files in which we're defining implicit methods, we can add an import clause:
   ```
   import scala.language.implicitConversions
   ```
2. Alternately, we can add a compiler flag:
   `-language:implicitConversions` ‡

‡ There are several other 'warning' flags; they can be disabled completely with a compiler flag: `-language:_`

Resolution of Implicits

In order to be applied, an implicit conversion must be *in scope*. There are two tiers of scope which Scala applies for resolution:

1. **Current Scope**: All the identifiers which are accessible without a prefix, i.e. those we could insert explicitly.
2. **Implicit Scope**: All of the members from the companion objects of *associated types*.

Implicit Resolution: Current Scope

The **Current Scope** is made up of all of the identifiers which can be accessed without a prefix, i.e. those we could insert explicitly. In order of precedence, they are:

1. Local Identifiers
2. Members of the enclosing scope(s), e.g. class or package
3. Imported Identifiers

Implicit Resolution: Implicit Scope

The **Implicit Scope** is made up of all of the members from the companion objects of *associated types*. In order of precedence, they are:

1. The type(s) in question, i.e. the source and possibly the target.
2. All the parts of a parameterized type, e.g. `A[B, C]`.
3. All the parts of a compound type, e.g. `A with B with C`.

Precedence of Scope

What happens if multiple matching implicit conversions are in scope? The compiler will apply **precedence** to select the most appropriate one:

1. Local identifiers
2. Members of an enclosing scope
3. Imported identifiers
4. The most specific member from the Implicit Scope

Precendence of Scope

Best Practices

Best Practice: To simplify the use of implicits, we should prefer the placement of implicit conversions inside companion objects.

- No special imports are needed.
- Local overrides are always possible, due to precedence.

“There Can Be Only One!”

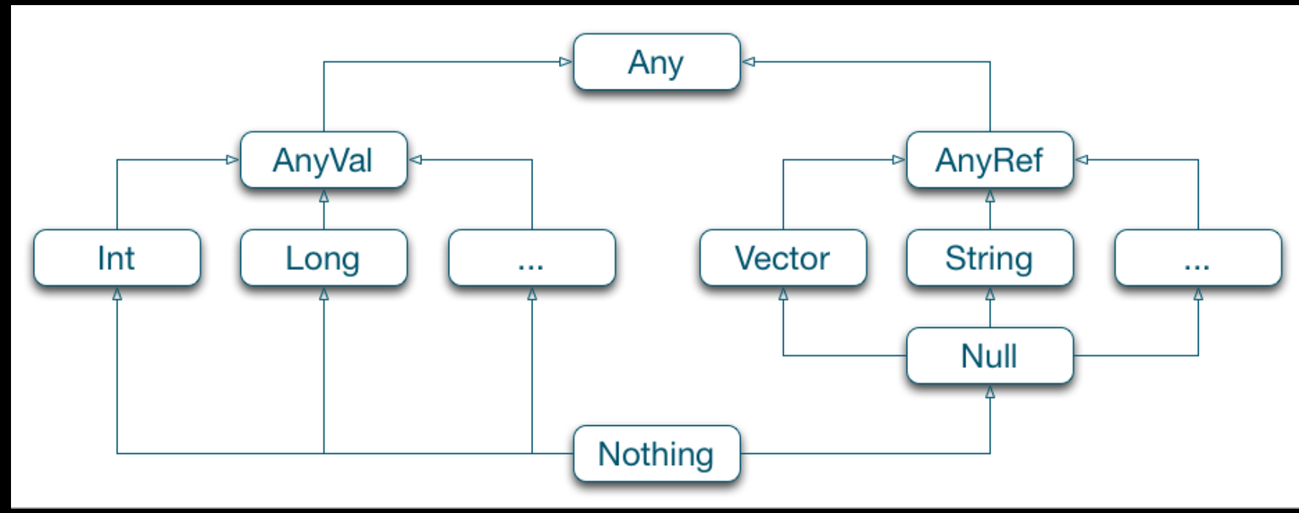
Multiple implicit conversions which share the highest located precedence are considered **ambiguous**, and will produce a compiler error:

```
implicit def stringToInt(s: String): Int = Integer parseInt s

implicit def stringToInt2(s: String): Int = Integer parseInt s

scala> "2" - 1
/*
<console>:11: error: type mismatch;
 found   : String("2")
 required: ?{def -(x$1: ? >: Int(1)): ?}
Note that implicit conversions are not applicable because they are ambiguous:
 both method stringToInt of type (s: String)Int
 and method stringToInt2 of type (s: String)Int
 are possible conversion functions from String("2") to ?{def -(x$1: ? >: Int(1)): ?}
*/
```

Digression: Custom Value Classes

Revisiting Scala's Type Hierarchy

- Normally, all user defined classes in Scala extend `AnyRef`, with `AnyVal` ("Value Classes") reserved for system primitives.
- Since Scala 2.10, we've been able to define custom Value Classes.
  - Value Classes wrap a single value.
  - All methods are inlined through the use of autoboxing.



Digression: Custom Value Classes

Writing Custom Value Classes

To define a custom Value Class, we must declare a new class which extends `AnyVal`:

```
class Weight(val underlying: Double) extends AnyVal {
  def +(that: Weight): Weight =
    new Weight(this.underlying + that.underlying)
}
```

Digression: Custom Value Classes

The Rules

There are also some rules we have to follow in defining Value Classes.

- They must accept *exactly* one class parameter.
- The single class parameter must be promoted to a `val`.
- There may not be any other fields (`val` and `var`) defined in the class.
- The Official Specification for Value Classes is defined in SIP 15*

---

* SIP is the Scala Improvement Process. Sort of like Java's JSRs or Python's PEPs.

Exercise #9: Implicit Conversions

Airport Codes as Objects

In AirScala, we often refer to case classes which are just a wrapper around `String`. It would be convenient if we could just say:

```
val ewrCode: AirportCode = "EWR"
```

Instead of:

```
val ewrCode: AirportCode = AirportCode("EWR")
```

- Let's retool `AirportCode` as a Value Class
- Next, in each companion object, create an implicit conversion of `String ⇒ <ValueClass>`
- Verify the provided tests pass.

Extension Classes

“Extend My Library”

Implicits allow us to extend classes – without subclassing – by creating a wrapper which defines new members, and defining an implicit conversion from the source to the wrapper:

```
class IntReverse(val n: Int) extends AnyVal {
  def reverse: Int =
    n.toString.reverse.toInt
}

implicit def intToIntReverse(n: Int): IntReverse =
  new IntReverse(n)

123.reverse
// res0: Int = 321
```

Best Practice: For performance reasons, use Value Classes with Extension Classes.

Implicit Classes

Scala 2.10 introduced improved syntax for extension classes, in the form of **Implicit Classes**. To define one, we prefix a class definition with the keyword `implicit`:

```
implicit class IntReverse(val n: Int) extends AnyVal {
  def reverse: Int =
    n.toString.reverse.toInt
}

123.reverse
// res0: Int = 321
```

- Similar to Value Classes, Implicit Classes must have *exactly one* class parameter.
- They must be contained in a package object, singleton, or class – methods cannot be top-level.

Exercise #10: Extend My Library

- The == operator is not typesafe, i.e. compares against Any
- Wouldn't a typesafe === operator be nice?

```
1 === 1 // true
"a" === "b" // false
1 === "1" // Won't compile
```

- Create the `Equal` singleton object in the `misc` package
- Create the implicit class `Equal0ps` in the `Equal` singleton object:
  - Add a polymorphic and typesafe === operator which delegates to the "natural" equality ==
- Play with your new code in the activator console (**HINT**: run `activator console` from your exercises directory, and `import misc._`)

Implicit Parameters

In addition to methods and classes, the `implicit` keyword can be used in method parameter lists:

```
def pow(x: Double)(implicit y: Double): Double =
  math.pow(x, y)
```

Implicit arguments can still be given *explicitly*:

```
pow(2)(3)
// res0: Double = 8.0
```

- **NOTE**: The `implicit` keyword may only be used in the *last* parameter list of a method.

Implicit Values

If implicit arguments are omitted, the Scala compiler will try to resolve them implicitly by looking for **implicit values**:

```
scala> pow(2)
// <console>:9: error: could not find implicit value for parameter y: Double
```

To define an implicit value, we use the `implicit` keyword:

```
implicit val defaultExponent: Double = 3.0

scala> pow(2)
// res0: Double = 8.0
```

- The standard implicit resolution rules apply to implicit parameters.

Implicit Parameters

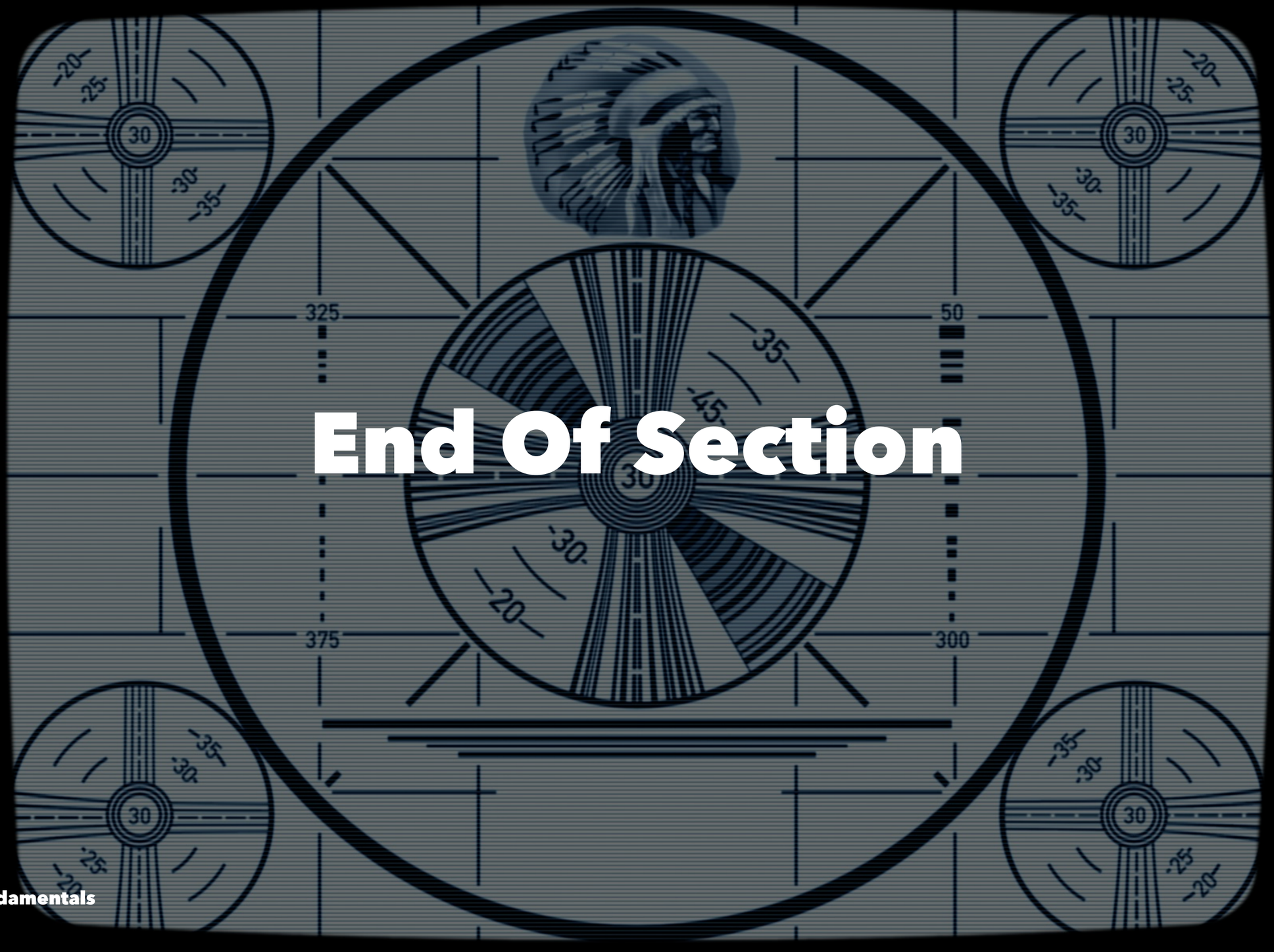
Use Cases

Implicit parameters give us support for **flexible defaults**; with them we provide default values, which the user can override. Alternately, we could combine implicit parameters with default ones:

```
def containerWeight(elements: Seq[Int])(implicit offset: Int = 0) =
  elements.sum + offset

containerWeight(elements)
// res0: Int = 6
implicit val offset: Int = 10
// offset: Int = 10
containerWeight(elements)
// res1: Int = 16
```

The Killer Feature: **Type Classes**[§]

[§] We will discuss Type Classes in a later section, but first: we need to perfect our knowledge of the Type System.


End Of Section